Introduction

AWS的收费模式：

1. AWS Lambda:按照function的执行时间以及cpu的使用情况
2. 工作流的额外收费：即当某个function的output成为另一个的input，这种function之间的调用需要产生额外收费
3. Greengrass：允许用户的数据交给edge处理，edge的收费按照VM的个数来收取。

problem:

edge的资源是有限的，但是可以同时运行多个方法，那么如何进行function分配才能使得Cost最少的同时而又不导致比之前更高的延迟?

本文主要技术点：

1. 提出AWS Lambda中不同的收费因素
2. 两种price model:
    i. price model for AWS lambda
    ii. execution time model————评估workflow中的执行时间以及通信消耗
3. 关于function聚合和迁移的算法

Background and Motivation

A. AWS Lambda Pricing

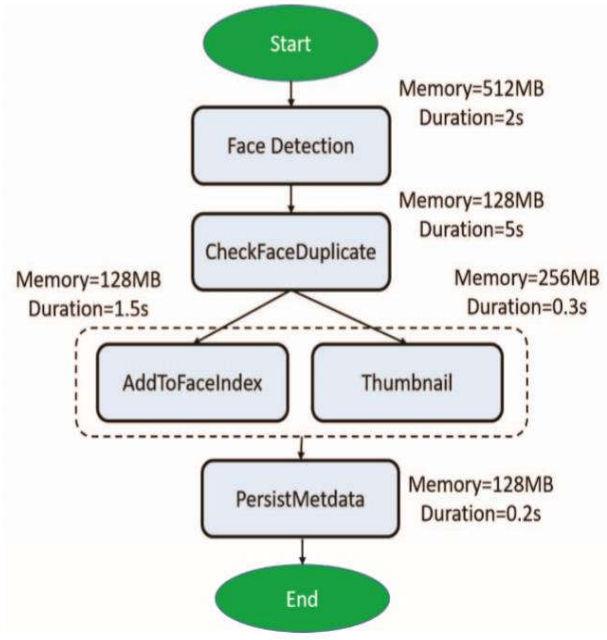

Fig. 1: Example AWS workflow (state machine)

如图，这是一个图像处理的状态机，它包含了五个funcrion。箭头表示的是数据依赖关系。

每个function的收费包括以下四方面：

1. 每个function每月的执行次数（比如1000000 excutions/month）
2. 分给function的内存。分配给function的cpu与其被分配的内存成正相关。（256MB的内存分配到的CPU将自动是128MB的两倍）
3. function的运行时间
4. 每1GB内存和每一秒的执行收费。AWS Lambda为0.00001667$/GB-s。

具体例子：如face detection，假设某个月执行了1000000次，那么总费用为：

$$Price_FaceDetection = 1000000execution * 512/1024GB * 2seconds * 0.00001667\$/GB - s = 16.67\$$$

除了上述的为每个function单独计费的项目外，**每个transition(function之间的数据传递)也需另外收费。**仍然按照1000000次进行计算，那么上图中的transition收费为：

$$Price_transition = 6transitions/execution * 1000000executions * 0.000025\$statetransitionprice = 150\$$$

综上总的费用（每个function的单独计费以及所有transition的费用）为： $35 + 150 = 185\$$

另外，**AWS Greengrass————作为连接edge和CLoud的服务，也需要另外收费**，每个edge device的费用大概为0.16-0.22。

B.Factors affecting price of serverless aplications

根据上面的收费模型，可以归纳出三个影响应用费用的因素：

1. state transition的个数
2. edge的使用情况
3. 内存的分配情况

根据这三点提出的优化：

1.state transition.

在构建应用的时候，把应用划分成多个function确实会提升扩展性和实用性，但是却会造成更多的数据依赖.

解决方法:**通过fusion来对大量function进行合并。**

存在的问题：

- **资源不匹配**

比如fig1中，如果把faceDetection和checkFaceDuplicate合并，那么在执行check部分的时候，仍然会采用512MB的内存运算5s，事实上此时只需要128MB。来看fusion前后的price对比：

$$P_fusion = 1000000 * 512/1024 * (5 + 2)seconds * 0.00001667\$ = 58.3\$$$

$$P_FaceDet + P_FaceCheck + P_transition = 1000000 * [512/1024 * 2seconds] + (128/1024 * 5) + 25 = 52.3\$$$

这种情况下fusion还不如之前存在transition的效果好。

- **并行的function**

比如AddToFaceIndex和Thumnail，如果其中一个或全部与前一个调用它的function进行fusion，那么本来的并行处理将会变成串行处理，从而导致更高的时间代价。

2.Edge vs Cloud Computation

Edge一般通过Amazon S3与Cloud进行通信（S3保存了Edge和cloud的中间数据），而Amazon S3与edge一样需要额外收费，大约0.023$/GB-month。

在本文的模型中，通过平衡计算速度与传输速率以及相关的price，可以使得cost和latency达到最优。

3.内存分配

AWS Lambda允许开发者自己为function分配内存，前面也提到了，cpu资源不能直接分配，cpu会随着内存分配自动成比例分配。而把function配置在128MB的VM或是256MB的VM上的过程是没有区别的，他们的区别在于运行时间。所以基于以上理论，本文接下来将针对不同内存大小的cloud VM进行测试，实验环境为：一个edge device和一个cloud。

Models and Problem Definition

A.Resource model

实验环境：一台edge设备，和一个云端。

变量定义：

- P_E :连接edge和cloud每月产生的费用。
- $f_i|i=1...n$ ： user的function集合。
- $m_{i,C}$:user为每个运行 f_i 容器或VM申请的内存大小。

假设：

- 只考虑内存这一种资源分配
- 所有function都在有限时间内完成
- 最大并发数为1000（每个function每次处理的请求不能超过1000）

B.Data Model

- 常规的数据采用JSON的格式进行封装
- 图片之类的二进制文件则需要先上传到持久层，然后把存储位置封装成JSON再交给Lambda function
- r :每月的请求数

C.Workflow model

对于AWS Lambda上的function，我们将他们视为一个有向连通图 $G_f=(V_f,E_f)$ ，其中节点代表每个function，而边表示数据依赖。

- $V_f=\{f_i|i=1...n\}$
- $E_f=\{f_i->f_j|i\neq j,i=1...n\}$

D.Function Profile

TABLE I: Table of Notation

n	=	Total number of functions
r	=	Total number of executions of a workflow
G_f	=	Input function graph
G'_f	=	Fused function graph
f_i	=	Function i in graph G_f
f'_i	=	Fused function i in graph G'_f
X_i	=	Placement variable (1: f_i on cloud, 0: f_i on edge)
t_i	=	Completion time of function i
$e_{i,C}$	=	Execution time of function i on the cloud
$e_{i,E}$	=	Execution time of function i on the edge
D_{fi}	=	Size (bytes) of output data of function f_i
$B_{E,C}$	=	Bandwidth (bytes/sec) between edge and cloud
$tr(E \xrightarrow{D_{fi}} C)$	=	Transmission time (sec) between edge and cloud
$s_{i,C}$	=	Time to schedule function i on the cloud
$m_{i,C}$	=	Memory allocated to function i
m_i	=	Maximum memory used by function i
p_E	=	Price of connecting one edge device to AWS cloud
p_s	=	Price of one state transition
$p_{m_{i,C}}$	=	Price of 1 sec exec. of function i with memory $m_{i,C}$
$P(G_f, X_{i=1,..,n})$	=	Price of workflow G_f according to $X_{i=1,..,n}$
$T(G_f, X_{i=1,..,n})$	=	Execution time of G_f according to $X_{i=1,..,n}$

E.Price Model

对于每个 f_i ，都对应于一个变量 X_i ，当后者为1时表示运行在Cloud，为0则表示运行在edge。总的价格计算如下:

$$P(G_f, X_{i=1,...,n}) = \sum_{i=1}^{i=n} X_i \cdot r \cdot e_{i,C} \cdot m_{i,C} \cdot P_{m_i,C} + r \cdot (n+1) \cdot P_s + P_E$$

说明：这里的计算模型和第二部分的例子一样，总的价格为所有function的云端费用、所有的transition费用以及一个edge的固定费用之和。

transition部分的n+1是因为start和end处也包括两次数据传递。

F.Execution Time Model

总的执行时间：最后一个function的结束时间减去第一个function的开始运行时间。

$$T(G_f, X_{i=1,..,n}) = t_n(G_f, X_{i=1,..,n}) - t_0$$

such that t_0 is the time before starting the execution of f_1. The completion time of function f_i is given by the recursive formula:

$$t_i(G_f, X_{i=1,..,n})) = \underbrace{t_{i-1}(G_f, X_{i=1,..,n})}_{\text{completion time of prev. function}} + \underbrace{(1 - X_i) \cdot e_{i,E}}_{\text{Execution time on edge}}$$
$$+ \underbrace{| X_i - X_{i-1} | \cdot tr(E \xrightarrow{D_{fi-1}} C)}_{\text{Transmission time}} + \underbrace{X_i \cdot (e_{i,C} + s_{i,C})}_{\text{Execution time on cloud}}$$

说明:

- Execution time on Edge：当 f_i 运行在云端时，(1- X_i)直接为0，否则为1。

- Transition time：($X_i - X_{i-1}$)意味着两个function只有同时在不同的运算设备（一个在edge一个在cloud）上才会存在transition。

G.Problem Definition

- G'_f ： fusion后的工作流图。
- $f'_i = f_1|f_2|f_3|...$ ： "|"表示连接符

优化问题为：

$$minimize : P_{G'_f, X_{i'=1,...m}}$$

$$staisfied : T_{G'_f, X_{i'=1,...m}} < T_{thresh}$$

说通俗点就是希望在function进行fusion以后，在产生用户可接受的延迟范围内尽可能的降低使用费用。

Proposed Approach

Fig. 3: Example Workflow with three functions

1. (f1 @ C)(f2 @ C)(f3 @ C)
2. (f1 @ C)(f2 @ C | f3@C)
3. (f1 @ C | f2 @ C)(f3@C)
4. (f1 @ C | f2 @ C | f3 @ C)
5. (f1 @ E)(f2 @ C)(f3 @ C)
6. (f1 @ E)(f2 @ C | f3 @ C)
7. (f1 @ E | f2 @ E) (f3 @ C)
8. (f1 @ E | f2 @ E | f3 @ E)

Fig. 4: Feasible function placement and function fusion solutions. Each line is one solution. Symbol "|" denotes fusion

这里给出一个构建"Cost Gragh"的具体事例，其中计算资源包括一个edge和一个Cloud，同时数据流向只能是edge流向cloud。
可行性方案： 每一个Solution需要解决的问题有两个，第一个是确定哪些function是需要被fuse的，同时要确定function fusion的具体运行位置（edge或cloud）。

如图四，为图三的一些可行性方案，其中”@C(E)“表示function的运行位置，” | “表示fuse操作，而一个括号内的function表示一个fusion。

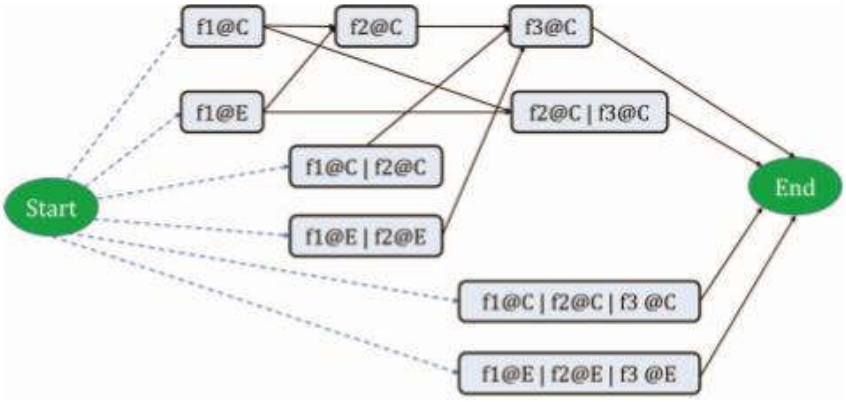

Fig. 5: Cost Graph, each path represent a solution for function placement and fusion. Each edge include execution time and price costs

代价图表示法 每一条边的含义包括两个独立的代价:

- 费用代价 c_{uv} :表示每个fusion运行产生的费用和fusion之间的数据传递产生的费用。
- 延迟代价 d_{uv} :表示每个fusion的执行时间和fusion之间的数据传递时间。

注意： 这里删除了很多不可性的方案，其中的限制因素包括：

1. f1->f2->f3的执行过程不可逆（所以不存在（(f2@C|f1@C)(f3@C)）。
2. edge->cloud的数据传递过程不可逆。

问题抽象：限制型最短路径问题。

问题定义：假设s和t是两个不同的节点，而s->t的所有路径集合为 Y_{st} ， s->t的任意一条路径用y表示。则有:

$$c(y) = \sum_{(u,v)\in y} c_{uv}$$

$$d(y) = \sum_{(u,v)\in y} d_{uv}$$

给定 $T_{thresh} > 0$ ， $\bar{Y}_{st}$ 表示满足 $d(y) \leq T_{thresh}$ 的y的集合。CSP(Constrained Shortest Path)的目的是从 $\bar{Y}_{st}$ 中找出一个使得c(y)最小的解，即 $y^* = argminc(y)|y \in \bar{Y}_{st}$ 。

Costless 算法步骤

step1:建立工作流表示类型

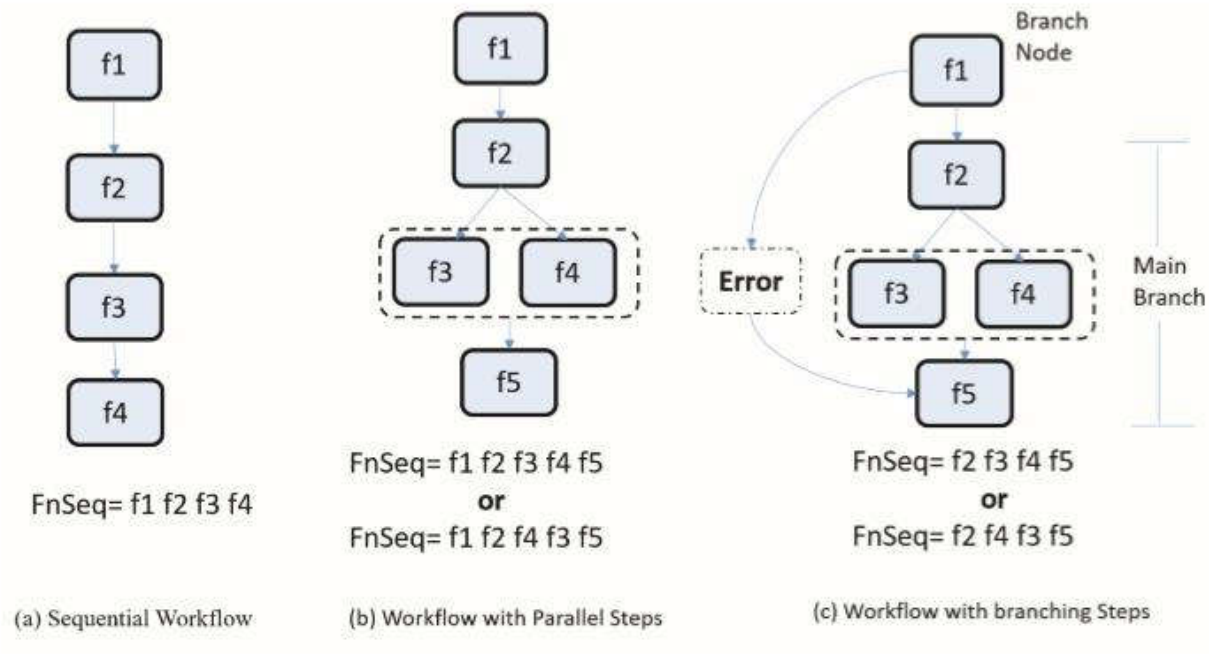

Fig. 6: Different workflow types supported by AWS

这一步的目的是确定好function的执行顺序。

step2:建立代价图

定义 $L(f')$ 表示f'可能的表现形式集合（edge或cloud），比如：

$$L(f' = (f_1)) = \{f_1@E, f_1@C\}$$
$$L(f' = (f_2\ f_3)) = \{(f_2@C \mid f_3@C)\}$$

同时考虑C上的两个内存，则有以下形式：

$$L(f' = (f_1\ f_2)) = \{(f_1@E \mid f_2@E), (f_1@C_{m_1} \mid f_2@C_{m_1}), (f_1@C_{m_2} \mid f_2@C_{m_2})\}$$

而L通过所有f'产生的所有可能的表现形式构成了代价图的节点。（如fig5）

step3:添加代价图的边

step4:解决CSP问题

NP完全。。。

近似解法——LARAC：核心思想就是用Dijkstra去找出 $c_{uv}/c^* + \lambda d_{uv}/d^*$ （这里是为了标准化，c*和*d都为最大值）这种混合代价的最短路径，而问题的关键在于找出最优的 λ 。

代价计算

以下面的路径为例:

$$(f_1@E) -> (f_2@C|f_3@C) -> (f_4@C) -> End$$

用T表示时间代价而P表示费用代价：

1) **Cost of link** $(f_1@E) \to (f_2@C \mid f_3@C)$:

$$T[(f_1@E) \to (f_2@C \mid f_3@C)] = \underbrace{e_{1,E}}_{\text{execution cost}} + \underbrace{tr(E \xrightarrow{D_{f_1}} C)}_{\text{transmission cost}}$$

$$P[(f_1@E) \to (f_2@C \mid f_3@C)] = \underbrace{p_E}_{\text{edge device price}} + \underbrace{r \cdot p_s}_{\text{transition price}}$$

If f_2 and f_3 are parallel:

$$T[(f_2@C \mid f_3@C) \to (f_4@C)] = \underbrace{max(s_{2,C} + e_{2,C}, s_{3,C} + e_{3,C})}_{\text{scheduling and execution time}}$$

$$P[(f_2@C \mid f_3@C) \to (f_4@C)] = \underbrace{r \cdot (e_{2,C} \cdot m_{2,C} \cdot p_{m_{2,C}}) + r \cdot (e_{3,C} \cdot m_{3,C} \cdot p_{m_{3,C}})}_{\text{functions price}} + \underbrace{2 \cdot r \cdot p_s}_{\text{transition price}}$$

if f_2 and f_3 are NOT parallel:

$$T[(f_2@C \mid f_3@C) \to (f_4@C)] = \underbrace{s_{2,C} + e_{2,C} + e_{3,C}}_{\text{scheduling and execution time}}$$

$$P[(f_2@C \mid f_3@C) \to (f_4@C)] = \underbrace{r \cdot (e_{2,C} + e_{3,C}) \cdot max(m_{2,C}, m_{3,C}) \cdot p_{max(m_{2,C}, m_{3,C})}}_{\text{fused function price}} + \underbrace{r \cdot p_s}_{\text{transition price}}$$

最后一条边的代价与第一条计算方法类似就不具体给出。

这里说明一下并行和串行的区别：并行因为需要同时执行，所以时间消耗只占一份而资源消耗需要占两份；而串行正好相反，相当于两个function合在一起上传并且在同一位置顺序执行，因此时间算两份，而费用代价只算一份。

算法分析

LARAC的复杂度： $O(|E^2|log^2(|E|))$ ，而|E|表示边的个数，其中 $|E| = |V \cdot deg_v$ ，|V|为点的个数， deg_v 表示v的出度。

m表示设备数量

下面进行复杂度计算:

$$|V| = \sum_{k=1}^{n}(n-k+1) \cdot m$$

其中k表示fusion的个数，n为function总数。

$$deg_v = (n-1) \cdot m$$

将上述两个等式带入|E|中：

$$|E| = m^2 \cdot (n-1) \cdot \sum_{k=1}^{n}(n-k+1) = m^2 \cdot (n-1)(n^2 - (n^2+2)/2 + n) = O(m^2 \cdot n^3)$$

$$O(|E^2|log^2(|E|)) = O(m^4 \cdot n^6 \cdot log^2(m^2 \cdot n^3))$$

Evluation

测试环境：

- edge设备：树莓派
- 云端：AWS Lambda
- 内存：默认分配为最接近每个functions所需的最大内存的允许内存（function最大所需100MB，实际选项只有96MB、128MB等，这时直接分配128MB）
- timeout：设置的足够大
- 假设数据都来自于树莓派并传到AWS S3然后交由云端处理。
- edge上的function最先执行然后将中间数据传给云

测试应用：Wild Rydes(交通应用类似于UBer)

> 用户需要上传照片进行注册，一旦上传完照片，就会执行fig7的工作流。f1为一个分支函数，这里不参与fusion，只考虑f2-f5

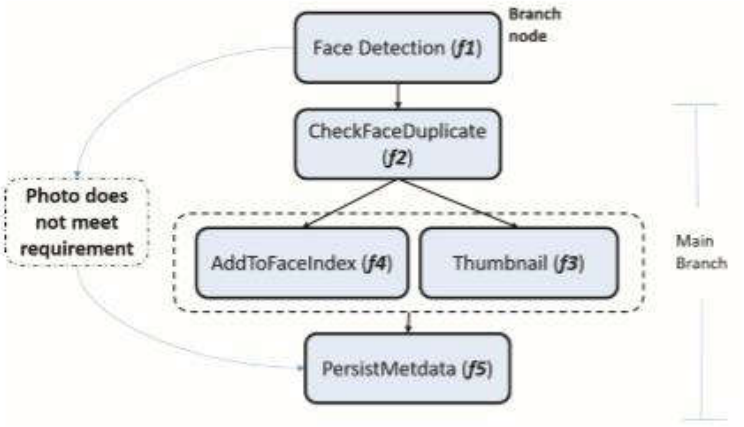

Fig. 7: Wild Rydes application workflow

应用剖析

> 通过让每个function在edge和cloud运行二十次测出平均值。考虑到每个function第一次运行都要进行container的初始化（为了之后复用），因此不把第一次放入评测。

Function	Avg. exec. time [128 MB / 256 MB / Edge]	Avg. scheduling delay	Max Memory used	Avg. billed duration [128 MB / 256 MB]
f1	893 ms / 772 ms / 1870 ms	61 ms	42 MB	955 ms / 822 ms
f2	970 ms / 743 ms	52 ms	38 MB	1016 ms / 800 ms
f3	2063 ms / 1080 ms	172 ms	83 MB	2116 ms/1144 ms
f4	844 ms / 735 ms	153 ms	37 MB	883 ms/788 ms
f5	153 ms / 101 ms	67 ms	38 MB	211 ms/144 ms

TABLE II: Profiling information for the functions in the Wild Rydes application (Figure 7).

评估标准

1. 模型准确率。

> 将执行时间与价格的评估值与观察值进行比较

2. 有延迟限制的价格

> 给定一个function gragh和一些deadline，比较costless和其他一些方法的结果，诸如暴力求解和一些启发式算法等。

3. 优化对内存配置的影响

> 对于每个融合和放置解决方案，我们在搜索每个函数的不同内存配置时显示价格优化。

4. 求解时间

评估结果

1. 模型准确率

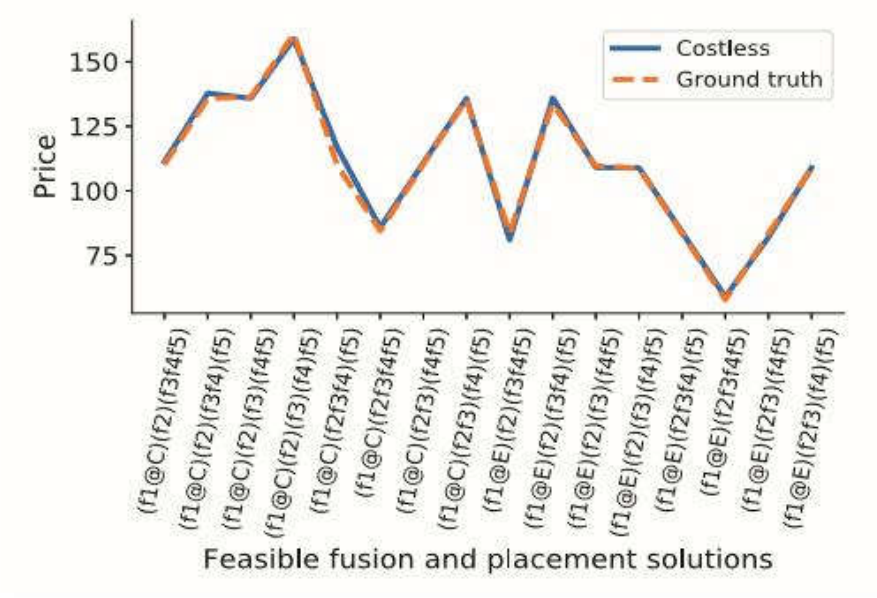

Fig. 8: Comparing pricing estimates of Costless with observed times during deployment of manually fused functions

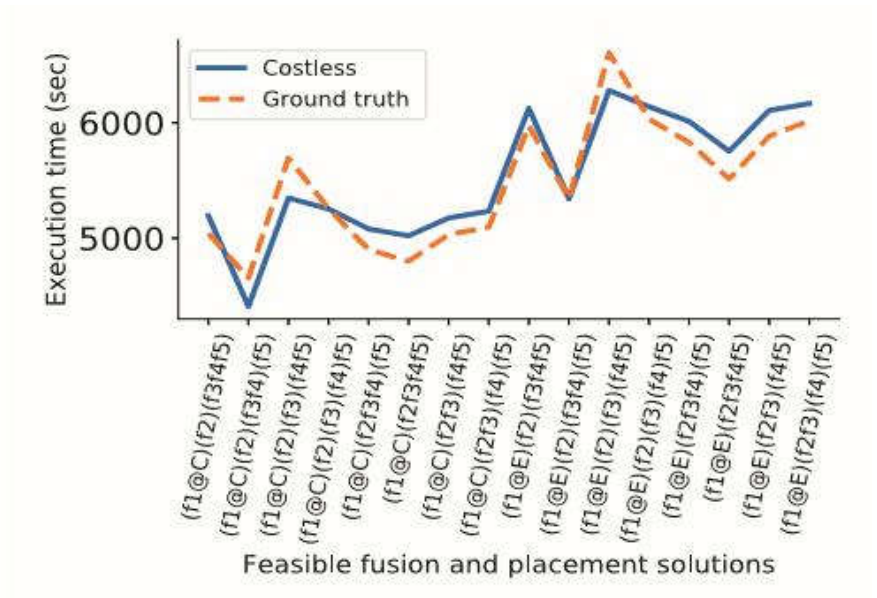

Fig. 9: Comparing execution time estimate of Costless with observed times during deployment of manually fused functions

> 执行时间的错误率为4%，费用评估的错误率为1.2%。执行时间的部分不吻合是因为网络延迟导致的。

2. 价格与执行时间的关系

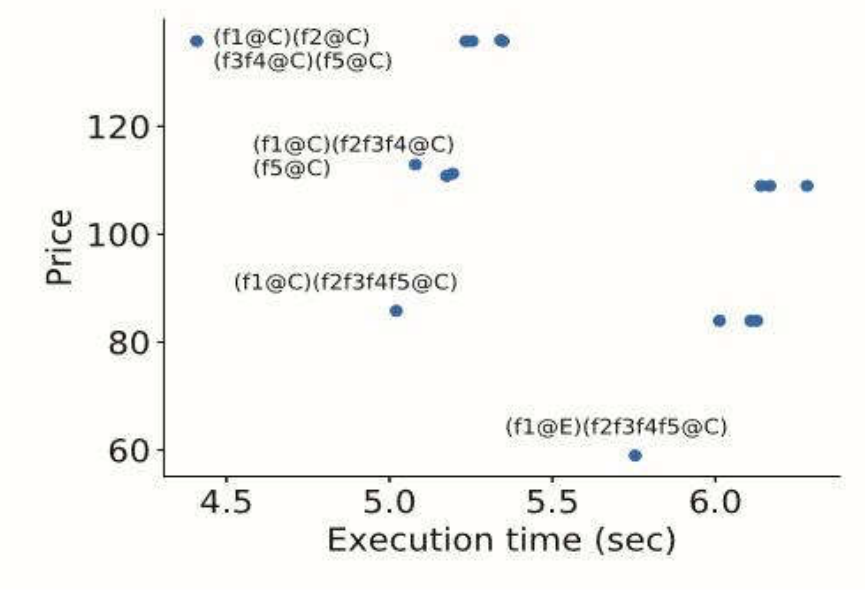

Fig. 10: Time and Price estimate for each feasible fusion and placement solution obtained by Costless

> 看不出明显的相关性，即执行时间并没有随着价格增长而降低。这是因为有些fusion会把两个并行function进行串行工作，这就导致时间增长而价格减少。 可以看到价格最高的那个肯定是初始grah（没有进行fusion的），而价格最低的那个是把f2-f5都进行fusion了，这不光会降低transition的费用还会大大降低调度延迟产生的运行费用。

3. 具有延迟约束的价格
比较几种价格优化策略：

- 真实值（对照组）——由AWS日志观察到的最优解。
- Costless
- 暴力求解
- Cloud(no fusion)

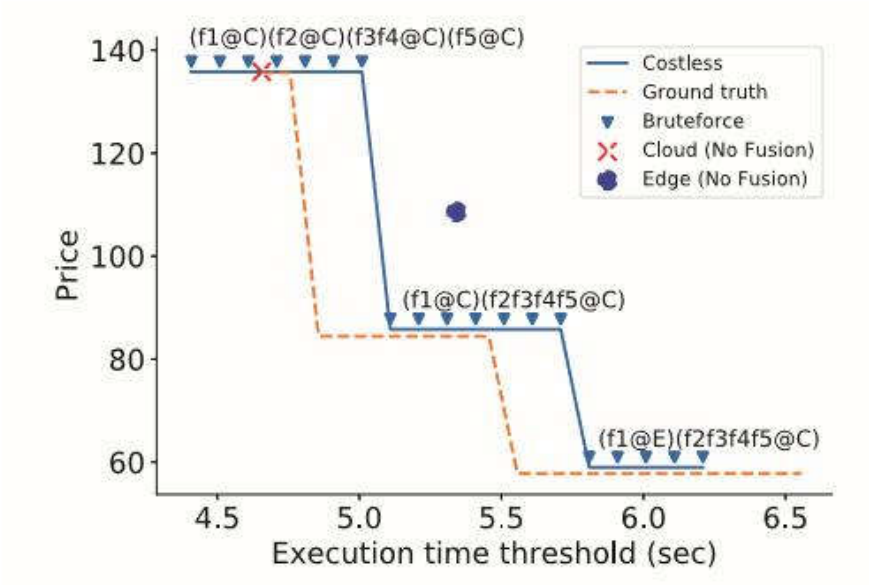

Fig. 11: Best price below an execution time threshold.

- Edge(no fusion)

> 可以发现，虽然Costless用的是近似解但还是和暴力求解得出的真实最优解效果近似。

Costless效果：

> 提高5%的延迟减少37%的费用；提高15%的延迟减少57%的费用

4. 不同内存环境的优化情况
   没看很懂。。。

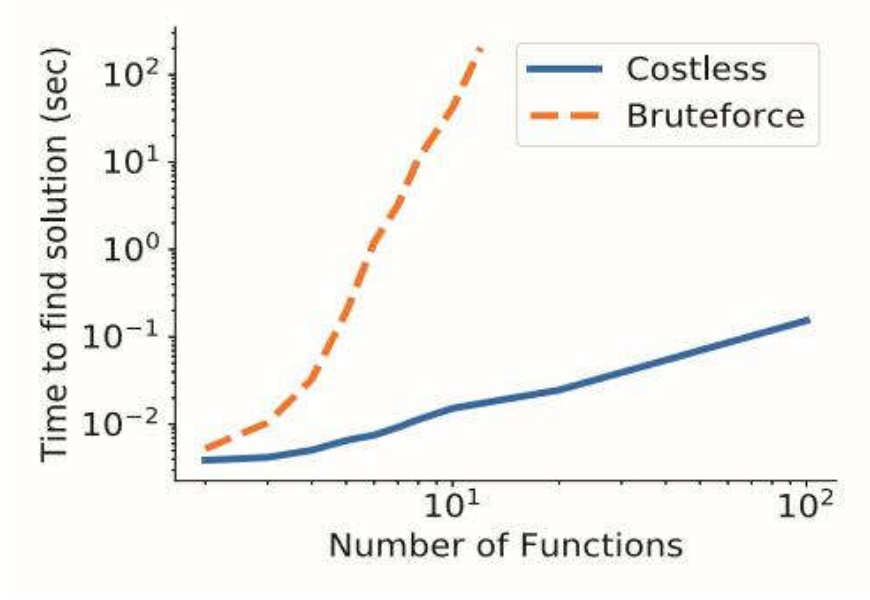

Fig. 13: Comparison between Costless and brute force in the time in the time to search for the best solution with increasing number of functions

5. 求解时间

> 比暴力法快很多，可以看到当function越多效果越明显。

Related Work